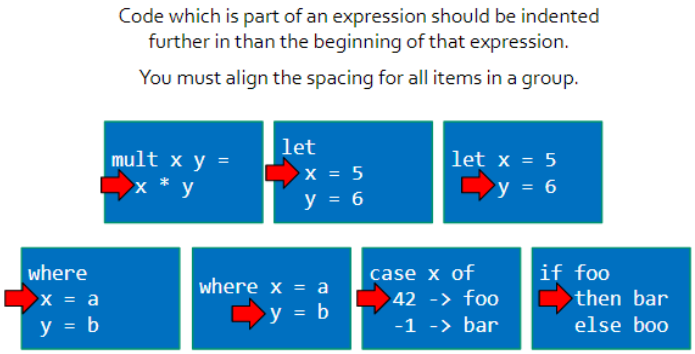
Functional Programming: Every func must take an arg, must return a val, must be “pure” and have no side effects, must be deterministic (same input x returns same output y). All vars are immutable. Funcs are like any other data and can be stored in vars and passed as args.
- **Pure** functions are deterministic and don’t mod data beyond initializing local vars required to compute output
FP:
func calls change things. Recursion - no loop or st8mnts, ez multithreading, order of execution is low importance, vs **Imperative**: statements, var changes, loops, func calls, multithreading is difficult and buggy, order of exctn is important

Haskell has **lazy evaluation** - doesn’t evaluate until value is NEEDED. Is statically typed and has **type inference**



Primitives: Int = 64-bit signed integers e.g., +/- 2^63. Integer=Arbitrary-precision signed integers. Bool = True or False. Char: ‘a’. Float - 32-bit, single-precision floating-point numbers (1.2E-38 to 3.4E+38). Double = Double-precision floating-point numbers (2.2E-308 to 1.8E+308)
/ division for Fractional. `**div**` and `**mod**` for Integral. Parens around neg numbers (-1)

3 composite data types:
Tuple - fst and snd only work on 2-tuples
List:. [Int] is a list of Ints. Implemented w Linked Lists. Const [**head**,...**tail**] = lst. **null** is the isEmpty function.
length, **take** n l returns list of first n elements. **Drop** n l returns list without first n elements.
lst !! i returns lst[i]. Zero-indexed. elem element lst checks if element is in lst. sum lst. product lst. **or** lst is like array.any(). **and** lst is array.all()
zip l1 l2 returns list of same length of 2-tuples ie [(l1[0],l2[0]),...] l1 ++ l2 concats two lists -> works on strings.
`elmt : lst` returns [elmt,...lst] (**cons operator**) **elmt1 : elmt2 : lst** works.
var :: type annotation
String = [Char]
Ranges are Inclusive
List Comprehension (HW1): lst = [(f x) | x <- input_list, (guard1 x), (guard2 x)]
lst = [(f x y) | x <- input_list, y <- input_list, (guard1 x), (guard2 x)]
For x in : for y in: f(x,y)

functions evaluated from left to right => left associative. Calls functions **BEFORE** operators

```
let
    gpa_part = 1 / (4.01 - gpa)
    study_part = study_hrs * 10
    nerd_score = gpa_part + study_part
in
    if nerd_score > 100 then
        "You are super nerdy!"
    else "You're a nerd poser."
```

Notice: nerd_score depends on gpa_part. This also works in where

Case expr of
    Const1 -> expr

```
        Const2 -> expr
        _ -> expr
```

Guards - HW1/2.
Pattern Matching - HW 1/2: f _ = 0. USE WHERE
expv2 :: (Int,Int) -> Int. expv2 (b,e) = b ^ 2. Can use _ to ignore/match anything: f (_, s) = s
f :: [Int] -> Int
f (first:rest) = first . THIS IS LIKE HEAD AND TAIL
f (first:second:rest) = first . THIS IS LIKE JS destructuring too
Can match length: f (first:[]) = “only 1elmt or maybe 0?” – just use f [] = default before this pattern

First-class functions - treated as any other data: can be stored in vars, passed as argos, returned by functions, and stored in Data structures.
A **higher-order function** is one that accepts another function as an argument or is a function that returns another function as its return value.
Mappers, Filters, Reducers,
Built In Haskell: **map** :: (a -> b) -> [a] -> [b] **filter** :: (a -> Bool) -> [a] -> [a]
Left associative:
foldl f accum [] = accum good for large but finigte lists, esp for commutative func like addition

```
foldl f accum (x:xs) =
  foldl f new_accum xs
 where new_accum = (f accum x)
```

Right associative

```
foldr f accum [] = accum
foldr f accum (x:xs) =
  f x (foldr f accum xs)
```

foldr can be used to process infinite lists, but foldl will crash
foldr (&&) True (repeat False) = False – *doesn’t converge for foldl*
Because with foldr the result has the form "f(x1, ...)", and x1 is immediately available.
With foldl the result has the form "f(..., xLast)", and xLast is not immediately available.
If lists were stored not as (first,rest) but as (rest,last), then foldl would be lazy and foldr would be not.
Lambdas = anon funcs (HW2). can assign these to variables
Identity func: (\x -> x).
(\x y -> x^3+y^2) 10 3 returns 1009

Capturing: slopeIntercept m b = (\x -> m*x + b). This is a **func generator**. M and b are captured, so they’re called “**free variables**” = any var that isn’t an explicit param to that lambda. Combo of a function with free vars and their values that were captured at the time of closure creation is a **closure**

Currying = convert func of mult args to func that takes first arg which returns func that takes second arg…
f(x,y) = (g(x)) (y)
F :: a->b->c->d === a-> (b-> (c->d))

```
def curried_f(): # f is x*y + z
        def a(x):
                def b(y):
                        def c(z):
                                Return x*y + z
                        Return c
                Return b
        return a  #   OR   return lambda p: lambda q: lambda r: p * q + r
```

Currying enables **Partial Function Application**
Algebraic Data Types HW2. | is OR
All Type Names and Variant Names must be capitalized. Breakfast, Lunch, Dinner and Fasting are variants of Meal

```
data Drink = Water | Coke | Sprite | Redbull
data Veggie = Broccoli | Lettuce | Tomato
data Protein = Eggs | Beef | Chicken | Beans

data Meal =
 Breakfast Drink Protein |
 Lunch Drink Protein Veggie |
 Dinner Drink Protein Protein Veggie |
 Fasting

careys_meal = Breakfast Redbull Eggs
pauls_meal = Lunch Water Chicken Broccoli
davids_meal = Fasting
```

The simplest Algebraic Data Type is just like a C++ enum.

We can also define more complex s one t).

David is fasting, so there are no food arguments to pass in.

And here's how we define several variables with our new ADTs!

data BSTree a = Nil | Node a (BSTree a) (BSTree a)

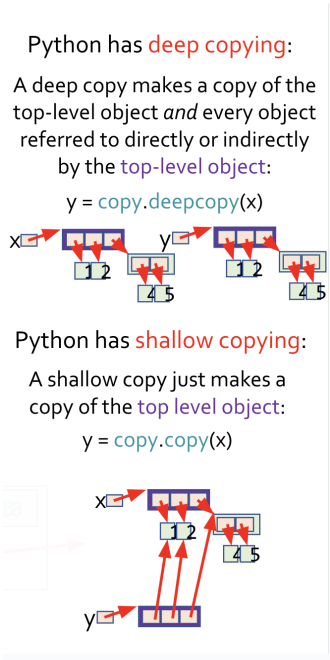
```
insert Nil v = Node v Nil Nil
insert (Node current left right) val
  | val == current =
     Node current left right
  | val < current =
     Node current (insert left val) right
  | val > current =
     Node current left (insert right val)
```

Currying- curry_f = return lambda p: lambda q: lambda r: p * q + r
Benefit of currying- every function with two or more parameters can be represented in curried form
i. a -> b -> c -> d
ii. (a -> b) -> c -> d
iii. a -> (b -> c) -> d
iv. a -> b -> (c -> d)
v. (a -> b) -> (c -> d)
vi. ((a -> b) -> c) -> d
vii. a -> (b -> (c -> d))
viii. a -> ((b -> c) -> d)
1,4,7 r the same| 6 is by itself |3 and 8 | 2 and 5 i, iv, vii| ii, v | iii, viii | vi

Python HW3
Strongly, Dynamically Typed. Function Scoped. Garbage Collection (w Reference counting, not on MT). Exceptions
Classes: no new keyword. Just c = Car(). Need to assign value to member vars. Cant just declare like c++.
__ prefix ~ private.
Object References ~ ptrs. For everything. Values are on heap

Class Method doesnt have self param. Cant access member vars, or call **instance methods**. They can access **class vars** and class methods
class x:
    class_count_this_is_a_class_var = 0
Called with x.class_count_this_is_a_class_var and x.class_method().

D = c; just makes a copy of pointer and not object.
__del__ is a finalizer that runs on GC. may never run.

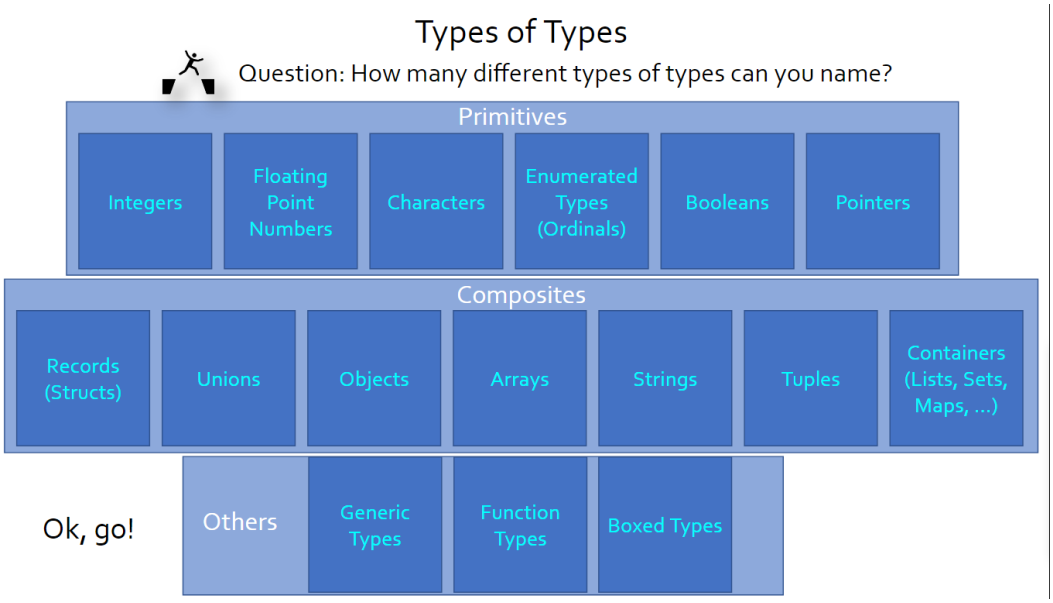


This is called "duck typing" – in Python, if an object has the requested method, a call to it will just work.

== is in __eq__. a is b checks if a and b refer to same obj
Id returns address of the VALUE ie address that pointer points to. Not address of where pointer is stored
Check **None**: x is None
isinstance(var, type): or **isinstance**(var, (type1,type2,...))m
Strings are immutable -> += causes it to point to a diff address. Even tho itsjust appending
Slicing[start:stop_exclusive:step]
Lists are mutable.
Tuples are immutable, ordered groups of items. Zero indexed. (1,2)[0] returns 0
Set s = set(). Add, remove, not in, {c for c in “a”} returns {‘a’}. - for set diff. | for union. & for intersection
Dict d = {}
In, del d[key]. For k,v in d.items(): for k in d:
Comprehensions: HW3
Lambda. **lambda** p: p + ‘asdf’ lambda p, x: p+x

Variable is a symbolic name associated w a storage location that contains a value or a ptr to a value
8 Facets:
Name - how u refer to var, type (var may or may not have an assigned type), value (value and its type), Binding - how a var name is connected to its curr val, its storage, its lifetime (timeframe over which a var exists), scope (where var name is visible to code), mutability
Value is a piece of data with a type, that is either referred to by a variable OR computed by a program expr
5 Facets:
Type (val always has a type). Value itself, storage, lifetime (timeframe over which the VALUE exists), mutability

Typed langs don’t necessarily require all vars to have a type.



Strong Typing = Language's type system guarantees that all ops r only invoked on objs/vals of appropriate types (else compile error or runtime). Ensures never have undef behavior at runtime due to type-related langs.
Reqs: Type-safe - prevent op on var X if X type doesn't support op. Memory-safe- prevents inappro mem access (out of bounds arrays, dangling ptrs). Checked Casts error if illegal cast. Pros: Less software vulnernabilities, earlier detection/fixing bugs
Basically every other lang.
Weak type checking = lang type system doesn't guarantee … Not Type safe, or memory safe. Undefined behavior at runtime bc of types. eg c++, c, assembly
Static Typing = b4 execution, type checker determines type of every expr and ensures all ops are compatible w the types of their operands. Lang MUST have a fixed type bound to each var at its time of definition. Type CANNOT be changed. Type Inference is possible.
Pros: Faster code bc no runtime type checks and optimizations r possible, detects bug earlier, no need for code to typecheck. Cons: is conservative and may errorout valid code. Requires type checking phase b4 execution - slow
C, C#, Go, Haskell, Java
Dynamic Typing - As program executes, type checker ensures each primitive op is invoked w values of correct types, or raises exception otherwise. Most langs don't require explicit type annotations for vars.
Types are associated w vals and not vars. Checked using type tags which r stored w vals.
Pros: Flexibility, easier to implement generics that operate on diff data types, simpler code bc fewer type annotations, faster prototyping. Cons: errors detected later, slower execution bc runtime check, requires more testing for same lvl of assurance, no way to guarantee safety across all possible executions.
Duck Typing only an option in dynamically typed langs.
Eg JS, Perl, PHP, Ruby, Python, Smalltalk

Gradual Typing - Some vars may have explicit types, some untyped. Type checking occurs partly b4 execution and partly during runtime. U can pass untyped var into typed var (runtime type check). Eg PHP, TS

C is a **subtype** of P iff 1. Every element in set of values of subtype C is also a member of set of values of sprtype P.
2. All ops that u can use on supertype must also work properly on subtype
Const int is supertype of int

Type **Conversion** = takes a value of type A and gens a whole new val (occupying new storage, w diff bit encoding) of type B. usually used to convert between primitives eg. float->int.
Float pi = 3.2; Cout << (int) pi;

Type **Casting** = takes a value of type A and treats it as if it was of type B - no conversion takes place. No new val is created. Usually used w objects. Eg below, int -> unsigned int.
Student m; // student is subclass of Person
Person& p = (Person&) m;

Explicit conversion/cast uses explicit syntax and tells compiler to do runtime check there. Implicit doesn't.
Explicit conversion: (int) pi; static_cast<int> pi; str(python_int); let x = 65 as char; parseInt(s);
Explicit cast: Student *s = dynamic_cast<Student*> (p); Student s = p as Student;
Implicit conversion = **Coercion.** Eg
Void foo(float f) {}; int main(){ int i = 42; foo(i); }
Coercion that converts subtype into supertype = **type promotion**
Implicit cast:
Void use_potty(Person* p){}; Nerd* n = new Nerd(); use_potty(n);
Nerd is subclass of Perosn. ^ is a **upcast**.
Downcast - typecast from superclass to subclass. Eg c++ dynamic_cast. This is dynamic type checking in a statically typed lang bc its at runtime.

Widening: subtype -> supertype. Widening casts/cnvrsns are **value-preserving**. Eg short -> int
Narrowing: supertype -> subtype, or between unrelated types. Eg float -> int

Checked in strongly typed langs
Some casts/cnvrsn may be **Unchecked** in weakly typed

Compilation vs Interpretation
We would expect the C-Lang executable to be faster. The C-Lang program is compiled directly to machine code, whereas the interpreted I-Lang code is translated into lower-level representations at run-time. This extra step creates additional overhead which will inevitably cause it to be slower than the pre-compiled code.
Exes aren't portable since they are compiled for specific ISA